REMARKS

Status

This Amendment is submitted in response to the Office Action of October 22, 2004 (hereinafter "the Office Action"). Upon entry of this Amendment, claims 8-26 will be canceled without prejudice, claim 1 amended, and new claims 27-38 added. Therefore, claims 1-6 and 27-38 will be pending, including newly submitted claims 27-38.

All references to the claims, except as noted, will be made with reference to the claim list above beginning on page 3. All references to "the Office Action," except as noted, will be referencing the most recent Office Action dated October 22, 2004. Line numbers in the Office Action, except as noted, will count every printed line, except the page header, but including section headings. Explanations of prior art references are based on the undersigned's best understanding thereof after careful review. If there is any confusion or questions regarding any aspect of this Amendment, the Examiner is invited to contact the undersigned.

Amendment

New claims are entered by this Amendment. Independent claims 27, 31, and 35 each set forth a method for optimizing loop structures in a computer program by reducing or eliminating index boundary checking. The disclosed method includes novel features now set forth in the claims including, as examples, sorting index expressions, building loop trees, and selecting compile time of a subroutine based on execution frequency. These features are supported in the written description as filed, e.g., in Figures 2, 3, 4, and 7, and associated text. Accordingly, no new matter has been entered by this Amendment.

Information Disclosure Statement

An Information Disclosure Statement is submitted along with this Amendment. The Information Disclosure Statement includes references cited in related U.S. Patent Application 09/872,456. In addition, U.S. Patent 6,343,375 (Gupta) is cited in accordance with Applicants' duty as set forth in 37 C.F.R. §§ 1.56 and 1.97. This reference was discovered upon entering in the USPTO database the authors' names of a reference cited in U.S. Patent 6,519,765 (Kawahito), col. 1, lines 58-61, which patent was cited in the most recent Office Action. Of possible interest is Figure 22 and related text, e.g., col. 23 in its entirety.

Rejection of claims 1-6 under 35 U.S.C. § 103(a)

Claims 1-6 stand rejected under 35 U.S.C. § 103(a) as being unpatentable over U.S. Patent 5,121,498 to Gilbert et al. (Gilbert) in view of U.S. Patent 5,668,999 to Gosling (Gosling). Applicants respectfully traverses because not all claim limitations are taught or suggested by the prior art of record and because the prior art lacks requisite motivation to combine the references.

Claim 1 sets forth a method for loop optimization including creating a loop structure having multiple loops for the purpose of reducing index range checking. Computer programs are written in a source code and are typically compiled into native executable code using compilers. Compilers are computer programs that convert the source code, which is understandable by humans, into executable code, which is generally understandable only by particular computer upon which the code is to be executed. In general, compilers have the capability to insert certain checks to trap errors, or bugs, in the program. One such check is known as a "array boundary check." The purpose of this check is ensure that an index to a data array does not point to a location in memory outside the array boundaries.

Claim 1 specifically sets forth, inter alia, "creating a pre-loop structure . . . capable of testing indexing expressions for underflow" (lines 11-13). With regard to this limitation, the Office action states, "Gilbert discloses a method for loop optimization within a dynamic compiler system, comprising: creating a pre-loop structure based on an original loop structure (3:20-24), for indexing expressions, see routine calls). . ." (page 2, lines 13-15). The portion of Gilbert referred to, presumably column 3, lines 20-24 states:

> The translator responds to a first loop instruction to unroll the loop into a set of straight-line instructions and retains a second-type loop instruction as a loop instruction. Those which are retained as loop instructions are thereafter processed as control code in the array master.

This text relates to a system described by Gilbert for executing program loops in a single-instruction multiple-data (SIMD) parallel computer system. This system includes an array master (AM) and an array of slave processors for executing code in a parallel fashion. The AM is a computer that provides an instruction source for the array. See column 4, lines 18-32. Note that the array referenced by Gilbert is not a data array as in the instant application, but instead an array of processors for executing code. This contrasts with claim 1,

which refer only data arrays, as is made clear by the context of the term and from the written description.

Figure 9 in Gilbert shows a diagram of a sequencer for the array master controller which provides specific hardware for handling loops, including a sequencer stack 74. When translating source code to executable code for the AM and slave processors, the translator has two methods for handling loops. These techniques are described by Gilbert in column 31, line 50 to column 32 line 27, and in particular, column 32, lines 12-19. In essence, depending on what instruction is inserted by the programmer, the translator will either maintain the loop as a loop or unroll it. Specifically, if the programmer uses the "LOOP" instruction, the loop will be maintained as a loop whereas if the programmer uses the "DO" instruction, the loop is unrolled in translation.

Thus, the text cited in the Office Action quoted above as relating to claim 1 actually refers to Gilbert's ability to translate program loops differently depending on the instruction used in the source code. If the LOOP instruction is used, then the AM processes the loop variable as control code as explained in Gilbert with reference to Figure 9. Thus, Gilbert really has nothing to do with reduction of boundary checking for array indexes. Furthermore, Gilbert has nothing to do with creating loop structures, other than to maintain a loop structure or unroll the loop structure.

Other limitations, which the Office Action suggests are shown by Gilbert are similarly absent. For example, the Office Action states that Gilbert discloses, "generating a main loop structure having indexing expressions based on the original loop structure, wherein the indexing expressions cannot produce an underflow (17:33 – 43, for main loop and original loop see nested loops). . ." (page 2, lines 15-17). The indicated paragraph does not mention indexing expressions for data arrays. An *expression* is a program phrase that returns a value. See Exhibit 1 beginning on page 15 of this Amendment. The index mentioned in the cited text actually refers to a stack pointer. Stacks are a type of memory data structure or buffer that are generally not accessible by index expressions but are rather accessed on a last-in, first-out basis. Underflow occurs when one tries to read data from an empty stack while overflow occurs when one tries to add data to a full stack. These concepts are different from the indexed data array described and claimed in the present invention wherein overflow occurs when the index is greater than the upper boundary of the array range and underflow occurs when the index is less than the lower boundary.

For example, if an array X is defined with a range from X[0] to X[100], then the compiler (or interpreter) will assign a certain amount of memory to that array. If and when an index falls outside of the range, i.e., below the lower boundary of 0 or above the upper boundary of 100, a danger exists that memory outside this defined memory space could be accessed or modified. For example, if a program attempted to access and/or modify X[105] in the above example, this could potentially cause harm to the program or system upon which the program is run. In contrast, stacks are generally not accessed using index expressions, so there are no boundary checks made other than to ensure that it is not empty when reading from the stack or full when writing to the stack.

Gilbert therefore does not teach "creating a pre-loop structure . . . capable of testing indexing expressions for underflow" and "creating a main loop structure . . . wherein the indexing expressions cannot produce an underflow." Gilbert instead teaches a SIMD system having particular features that enable fast execution of program loops using an array of processors.

Gosling does not correct the deficiencies of Gilbert. The Office Action, page 2, lines 18-20 states that, "Gilbert doesn't explicitly disclose wherein the pre-loop structure is capable of testing indexing expressions for underflow and wherein the post-loop structure is capable of testing indexing expressions for overflow. However, Gosling does disclose this feature in an analogous art (FIGURE 4C, 452, FIGURE 4D, 472, also related text 5: 5 –10, and 8: 40 – 50)." Again, the portions referred to in Gosling only relate to stacks, which, as described above, are not accessed using index expressions. Claim 1 requires index expressions. Thus, neither Gilbert nor Gosling teach or suggest that which is set forth in claim 1.

Gilbert and Gosling are improperly combined by the Office Action. Specifically, the Office Action states, "it would have been obvious . . . to combine Gilbert and Gosling because, using different iterations of the loop to perform different functions makes the program more efficient," (page 2, lines 22-25). Applicants respectfully request clarification of this statement. Which reference, and where in the reference, is it suggested that different iterations of the loop perform different functions? In what way does this make it more efficient? How, exactly are the references combined? Applicants respectfully submit that the Office Action has not provided sufficient basis or explanation of any motivation in the prior art to combine the references sufficient to present a prima facie case of obviousness. The Office Action does not point to where, in either reference, it is taught that different iterations

of the loop perform different functions, or how that would make the program (which program?) more efficient.

Applicants respectfully submit that claim 1 is allowable over the references cited for the reasons set forth above. Furthermore, Applicants submit that, since claims 2-6 depend from claim 1, they should be allowed for at least the same reasons as claim 1. Applicants therefore respectfully request withdrawal of the rejection against claims 1-6.

Rejection of claims 9-20 under 35 U.S.C. § 103(a)

Claims 7 and 9-20 stand rejected under 35 U.S.C. § 103(a) as being unpatentable over Gilbert in view of Gosling, and further in view of U.S. Patent 6,519,765 to Kawahito et al. (Kawahito). Applicants respectfully traverse because the rejected claims have been canceled. However, Applicants will take this opportunity to address Kawahito.

Kawahito discloses a method for eliminating redundant array range checks in a compiler. The Office Action specifically points to column 3, lines 50-56 in Kawahito (Office Action, lines 13-15). This section of Kawahito refers to a technique referred to by Kawahito as "versioning." Versioning is best described by Kawahito in column 15, lines 52-65. In essence, when range check elimination cannot be performed for an entire loop, two versions are performed, and code is generated such that range-check code is generated at the beginning of the loop, and code for loop processing which does not require array range checks and a code for loop processing which requires them are respectively generated. Depending on the result of the range check expression, one or the other loop is executed. See, e.g., Table 7 (column 11) which shows a range check in line 3, a first loop in lines 6-10 and a second loop in lines 14-21. This is different from the presently disclosed invention wherein the generated loops are not alternatively executed, but serially executed. In other words, the versioning provides two versions of the loop, and the version executed depends on conditions upon entering the loop whereas the presently disclosed invention involves creating separate loops for different ranges of the index variable so that each are executed in turn. This technique is referred to as "iteration splitting," a term defined on page 17, lines 10-11 of the specification and set forth, for example, in claim 1, line 8.

Response to Arguments

Page 5, lines 21-24 of the Office Action addresses whether Gosling teaches "testing indexing expressions for underflow," and points to column 5, lines 5-10 and columns 8, lines 40-50. However, while Gosling teaches checking a stack pointer (or index) for underflow, this is not the same thing as testing an indexing expression for underflow. See appended hereto Exhibit 1, wherein the term, "expression" is defined as, "a program phrase which returns a value." The stack pointer is not a program phrase, i.e., a phrase in the program. It is a value stored in a register for accessing a stack internally. Because stacks are not normally accessed using indexes, and are generally accessed only on a first-in, last-out basis, there would be no suggestion to provide an expression returning the value of the stack pointer. Furthermore, even if one did provide an expression returning the value of the stack pointer for the purpose of reading the stack data, it would not be tested for underflow because Gosling only teaches checking for underflow when there is an instruction that pops data from the stack (removing the top most item(s). See column 8, lines 42-45: "If the currently selected instruction pops data from the stack (450), the stack counter is inspected (452) to determine whether there is sufficient data in the stack to satisfy the data pop requirements of the instruction." Because there is no index expression in the pop statement (which only requires the top most data of the stack to be read) Gosling does not teach testing an indexing expression for underflow.

As amended, each of the independent claims set forth, *inter alia*, sorting the index expressions by the trip counter and offset. See claim 1, lines 6-7; claim 27, lines 6-7; claim 31, lines 10-11; and claim 35, lines 5-6. None of the references of record, either alone or in combination, teach or suggest this feature. For at least this reason, therefore, Applicants respectfully submit that all pending claims are allowable. A Notice of Allowance is therefore respectfully requested.

If the Examiner has any questions concerning the present amendment, the Examiner is kindly requested to contact the undersigned at (408) 749-6900 X6933. If any other fees are due in connection with filing this amendment, the Commissioner is also authorized to charge

Deposit Account No. 50-0805 (Order No. SUNMP017). A duplicate copy of the transmittal is enclosed for this purpose.

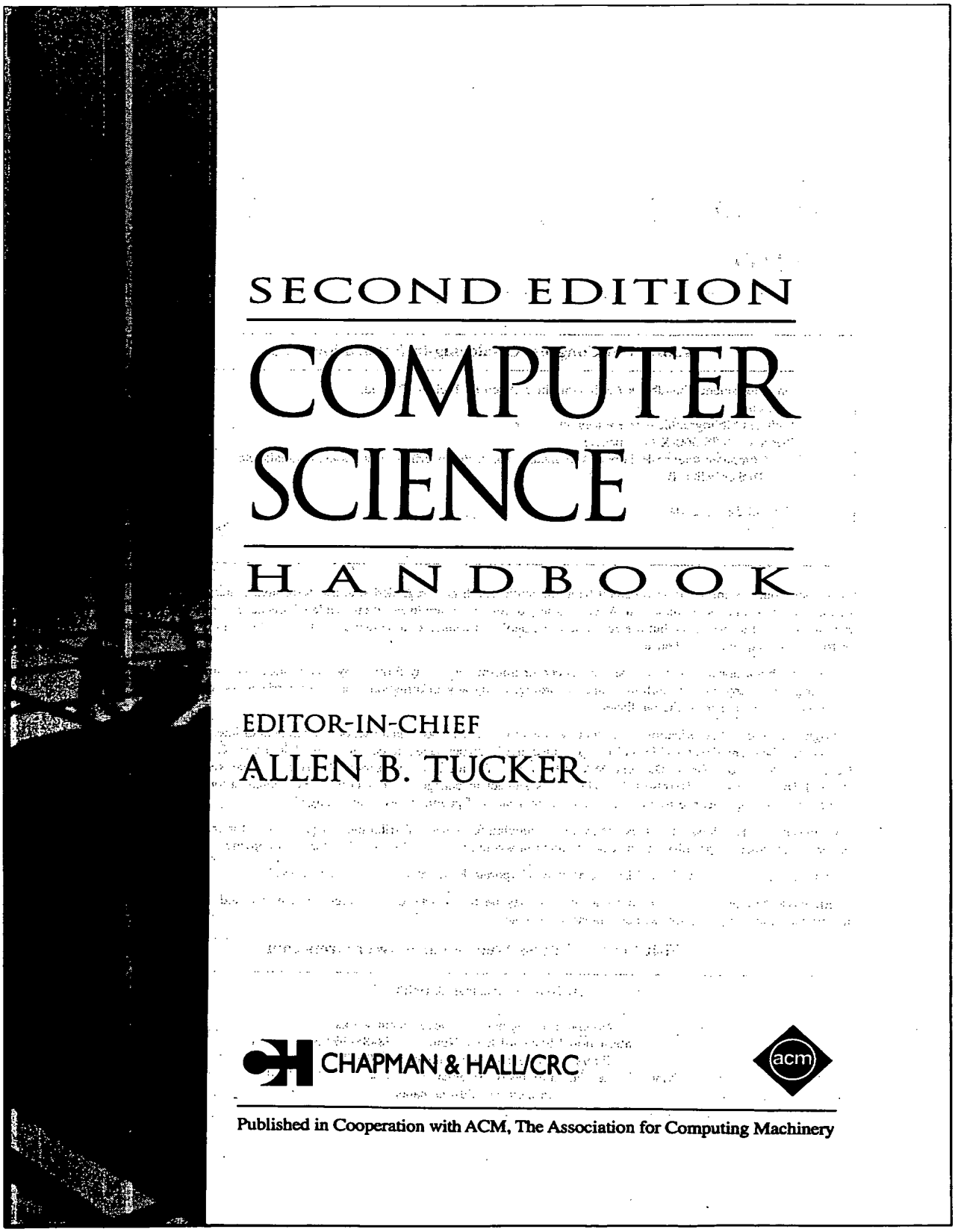
Respectfully submitted,
MARTINE & PENILLA, LLP

Leonard Heyman, Esq.
Reg. No. 40, 418

710 Lakeway Drive, Suite 200
Sunnyvale, CA 94085
Telephone: (408) 749-6900
Facsimile: (408) 749-6901

Customer Number 32291

EXHIBIT 1:


SECOND EDITION
COMPUTER SCIENCE
HANDBOOK
EDITOR-IN-CHIEF
ALLEN B. TUCKER
CHAPMAN & HALL/CRC
acm
Published in Cooperation with ACM, The Association for Computing Machinery

Library of Congress Cataloging-in-Publication Data

Computer science handbook / editor-in-chief, Allen B. Tucker—2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 1-58488-360-X (alk. paper)
1.Computer science-Handbooks, manuals, etc. 2. Engineering—Hanbooks, manuals, etc.
I. Tucker, Allen B.

QA76.C54755 2004
004—dc22 2003068758

International Standard Book Number 1-58488-360-X
Library of Congress Card Number 2003068758
Printed in the United States of America 1 2 3 4 5 6 7 8 9 0
Printed on acid-free paper

```
with TEXT_IO; use TEXT_IO;
procedure SCOPED is
  package INT_IO is new INTEGER_IO (integer); use INT_IO;
  I,J: integer;

  procedure P is begin put (J); new_line; end P;

  begin
    J := 0;
    I := 10;
    declare   -- Block 1
      J: integer;
    begin
      j := I; -- reference point A
      P;
    end;
    put (J); new_line;
    declare   -- Block 2
      I: Integer
    begin
      I := 5
      J := I + 1; -- reference point B
      P;
    end;
    put (J); new_line;
  end;
```

FIGURE 90.1 Scoping rules in Ada.

As an example of scope rules in Ada, consider the code in Figure 90.1. Static scope rules are determined by the program block structure, which does not change while the program runs. Therefore, the call to procedure P prints the variable J defined in the outer, main program, no matter where it is called from. Likewise, the assignment in block 1 at reference point A changes J from the block and not from the main program. Dynamic scope rules, on the other hand, typically follow dynamic call paths to determine variable bindings. If Ada used dynamic scope rules, the first call to P from block 1 would print the value 10 corresponding to the J from block 1, whereas the second call to P would print the value 3 corresponding to the J from the main program.

90.2.5 Execution Units: Expressions, Statements, Blocks, and Programs

An *expression* is a program phrase which returns a value. Expressions are built up from constants and variables using operators. As described earlier, variables may represent two values, depending on context: their location and the value stored at that location. Operators may be builtin, like the arithmetic and comparison operators, or may be user-defined functions.

Reflecting the sequential order of von Neumann computation, an imperative language specifies the order in which operations are evaluated. Typically, evaluation order is determined by precedence rules. A typical precedence rule set for arithmetic expressions might be the following:

1. Subexpressions inside parentheses are evaluated first (according to the precedence rules).
2. Instances of unary negation are evaluated next.
3. Then, multiplication (*) and division (/) operators are evaluated in left to right order.
4. Finally, addition (+) and subtraction (−) are evaluated left to right.